MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs

Randy Osborne

TR94-02c December 1994

Abstract

This paper presents a new, ẗhybrid depositm̈odel for low overhead communication wherein the sender directly deposits messages into the destination user-level memory. The destination address is a function of both sender state and destination state. The motivation is to increase the senderś role in communication in order to simplify the destinationś role and thus enable fast, low-cost communication interfaces. The model separates data delivery from synchronization so as to enable the optimization of simple data delivery while leaving more difficult synchronization to other mechanisms. With hardware support, the hybrid deposit model looks promising for applications in parallel, distributed, and real-time computing.

Fourth International IFIP Workshop on Protocols for High-speed Networks, August 1994

1. First printing (version 1), March 24, 1994
2. Second printing (version 2; additions and revisions), April 3, 1994
3. Third printing (version 3; revisions for publication), June 28, 1994

1 Introduction

The considerable recent work on workstation network interfaces for high speed LANs (e.g. [Dav93, BP93, TS93]) has focussed on achieving high bandwidth. This paper focuses, in contrast, on low overhead communication — by which we mean both low latency and low impact on the host processor — for high speed LANs. We present a low level protocol and network interface architecture for low overhead communication. The key idea of this paper is to use both sender information and destination information to demultiplex messages directly to where they are needed. We show that this combination has flexibility for a wide range of application requirements.

We have built a software implementation to evaluate these ideas. We get a best case latency for a 32 byte application to application data transfer of 24.5μsec on DECStation 5000s connected via an ATM network (*sans* switch; 30μsec with an ATM switch). This is about 10 times faster than the fastest conventional approach (using Fore Systems' AAL3/4 implementation) on the same hardware[Ke94]. With appropriate hardware support, we believe latencies of under 10μsec for a 155Mbps ATM LAN and under 3μsec for a 622Mbps ATM LAN are possible in the workstation LAN environment.[1]

Our interest in low overhead communication is motivated by applications in parallel, distributed, and real-time computing. Latency, one of the fundamental issues in parallel computing [AI87], limits the maximum parallelism that can be exploited in an application. In distributed computing, low latency can help improve the performance of remote procedure call (RPC) and thereby enhance the performance of the ubiquitous client-server model. Low latency can also be useful for real-time computing, but far more important is predictability: it is important to insulate real-time tasks on the host processor from unrelated asynchronous communication events. Finally, low latency communication increases the flexibility in structuring a system.

Our goal is to achieve minimal communication latencies between application processes across high speed LANs. We assume the compute nodes are multiuser with their own address spaces and memory (e.g. workstations) and we assume the network is multiuser, so there must be protection against accidental or malicious interaction of users. It is important that the communication allow random access data transfers for parallel computing.

Section 2 describes the problem and related work. Section 3 presents a "hybrid deposit" communication model that uses both sender and destination information, maintaining full multiuser protection. Section 4 specializes this hybrid deposit model to ATM LANs. Section 5 describes a software implementation of this model. Section 6 presents an interface architecture called DART. Finally, Section 7 concludes and discusses further work.

[1] For one transit of a fast ATM switch.

	Data delivery	Interrupts
Destination-based	$address_{msg} = f(dest)$	$intr_{msg} =$ always
Sender-based	$address_{msg} = f'(sender)$	$intr_{msg} = g(sender)$
Hybrid	$address_{msg} = f''(sender, dest)$	$intr_{msg} = g'(sender, dest)$

Table 1: Message Passing Options

2 The Problem and Related Work

The main problem to achieving low overhead communication is how to handle asynchronous message arrivals at the destination to meet all four requirements of low latency, low impact, low cost, and multiuser protection simultaneously. (Sending is easier because it's synchronous — to a first approximation.) Many global address space parallel machines sacrifice protection to get a very cheap solution in which messages write directly into destination memory. Most workstations sacrifice latency and impact to get a low cost solution by multiplexing application and message processing on a single processor. Consequently, an asynchronous message arrival incurs overhead in *both* the communication and whatever application happens to be running at the time. High performance systems sacrifice cost and duplicate resources to bypass the host processor and operating system e.g. parallel machines such as the Intel Paragon and *T [NPA92] and real-time systems such as Spring OS [SR91]. These systems devote hardware for the worst case requirements.

An intermediate solution is to control asynchronous events by separating events by their need for the host processor. Events, such as data delivery, that don't require the host processor can be handled directly e.g. by depositing data directly into user memory. Events — chiefly synchronization — that do require the host processor can be divided into immediate actions that require immediate service and delayable actions that can be accumulated and processed when convenient for the host processor (thereby turning them into synchronous events). With this separation of data and control events we only need resources sufficient to bypass the non-host processor events rather than all events.

Conventional communication protocols (especially for LANs) are not designed to facilitate fast separation of incoming messages into these various event types, increasing the processing required at the destination. Instead, we can exploit the sender's knowledge, shifting more of the burden to the sender, in order to simplify the message processing at the destination. In effect, the sender uses its knowledge to pre-demultiplex the messages. By doing so, we posit three benefits: greater accuracy in separating events at the destination, simpler and cheaper communication co-processors, and reduced host processor load.

Table 1 shows a classification of messaging based on the division of the burden between sender and destination. *dest* represents destination state, *sender* represents sender state transmitted with the message, and $intr_{msg}$ is the interrupt status on message arrival. In *sender-based addressing* demultiplexing is trivial: a message contains an address, supplied by the source, into which the message is directly deposited. In *destination-based addressing*, the source has no direct input on the final address of a message: a message identifies a buffer into

which the message is stored at some implicit location, e.g. by sequencing a pointer.

Destination-based addressing is the conventional approach in distributed systems and for send-receive models in parallel machines. Sender-based addressing is common in parallel machines, though usually at fine granularities e.g. words. Recently, sender-based addressing has become popular for optimizing messaging passing, as in SHRIMP [Be94] which uses virtual memory mapped communication. Three recent works promote sender-based addressing in a LAN environment. [Wil92] describes a high level design of an interface called Hamlyn; [SP90] describes an interface design called Axon; and [TLL93] describes a in-kernel implementation of a remote read/write model (a follow-on of Spector's work [Spe82]). The Meiko CS-2 multicomputer also supports a remote read/write model, though with custom co-processors and proprietary network.

Sender-based addressing allows random access transfers and thus is attractive for parallel computing. However, sender-based addressing suffers from two problems in our context of a LAN distributed system. The first problem is lack of isolation. Pure sender-based addressing releases considerable information to the sender and requires the destination to trust the sender. The second problem is the inefficiency of certain operations. For example, three messages are required to add to the end of a shared queue. Whereas this is acceptable in a fine grained global address parallel machine, it is not acceptable in a LAN environment with its higher communication costs.

Hybrid-based addressing solves these problems while retaining the advantage of sender-based addressing. The storage address is partially a function of an address the sender may not know, either for isolation, or so we can add to the end of a shared queue in one message. Active Messages provide hybrid addressing. An "active message" contains the name of an interrupt handler, arguments, and data. The destination executes the interrupt handler to apply the message action [von92]. This provides great flexibility in combining sender and destination information. However, to get low latency these interrupt handlers execute on the host processor and execute in the context of the interrupted task. Thus Active Messages must be combined with a protection mechanism to be useful in a multiuser environment. [DLM94] provides a limited form of hybrid addressing in which a message controls whether it is deposited at an address contained in the message or an address stored at the destination.

Hybrid-based interrupts are also useful. Pure destination-based interrupts, like in Active Messages, cause an interrupt on every message arrival. On the other hand, pure sender-based interrupts, as used in Hamlyn, limit the interrupt semantics: it becomes very awkward, for instance, to priority schedule interrupts at the destination. Thus, as with addressing, interrupts should be a function of both sender and destination information. This allows the interrupt status to be partially a function of message priorities contributed by processes that the sender may not know exist. [TLL93] and [DLM94] describe very limited approximations to hybrid interrupts in which the sender indicates an interrupt in the message and the destination can simply enable or disable all interrupts for a given buffer.

Our work combines sender and destination information for both addressing and interrupts. This gives flexibility to exploit various points in the sender-destination spectrum, but with full protection unlike with Active Messages. This full hybrid functionality, for both data and

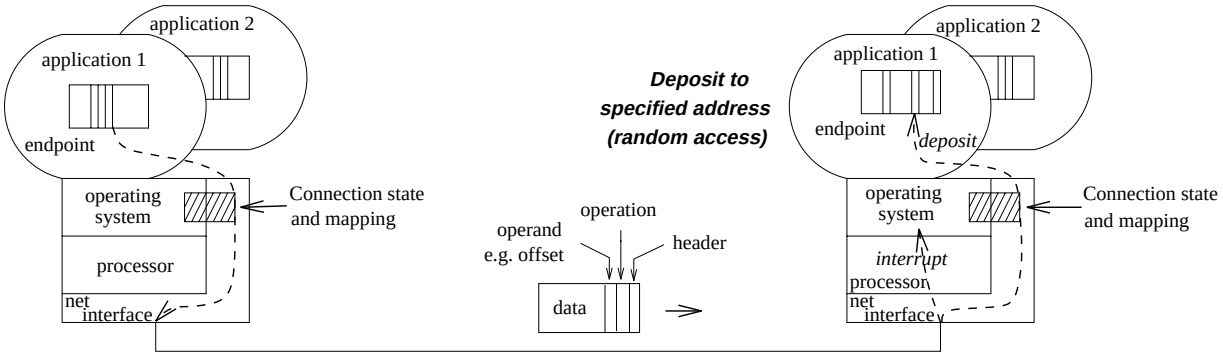

Figure 1: Hybrid deposit operational model

control, distinguishes our work from previous work.

3 Hybrid Deposit Messaging

[Se93] introduced the "deposit model" for a form of sender-based addressing used by a compiler in a message passing parallel computing paradigm. We generalize this appropriately suggestive term to mean demultiplexing messages and depositing them directly where they are needed e.g. depositing data directly in application memory and delivering (significant) control events to the host processor (similar to the ideas in [Se94]). To this notion we add hybrid addressing and interrupt generation.

3.1 Overview

The application view of hybrid deposit messaging is two "endpoint" buffers linked by a "connection". As shown in Figure 1, messages originating in the source endpoint can bypass the conventional operating system and host processor route to the network and either be delivered directly to any location in the destination endpoint or conditionally delivered to the destination operating system. Messages contain both control and data information. The destination decodes the control information and chooses an action and/or destination location based on **both** the source state contained in the message and destination state. Although these actions may be quite general, we focus on a set of primitive actions representing common operations that can be implemented simply without host processor or operating system intervention. We leave more complex actions to the host processor. This is the RISC philosophy of processor architecture applied to communication.

3.2 Endpoints and Connections

Endpoints and connections are allocated and deallocated with kernel calls. An endpoint (buffer) is any page-aligned, contiguous region of virtual memory. Consequently, the host

virtual memory page protection can be used within endpoints. A connection is a channel authorizing communication between a pair of endpoints (connections can also be duplex and multicast). Some state and protection information is associated with the source and destination end of each connection.

An endpoint may have multiple originating connections and/or multiple terminating connections. All connections associated with a given endpoint use the same virtual memory mapping and protection information. Endpoint buffers can be overlapped or nested to effect different degrees of sharing and isolation between connections (like fbufs [DP93]). Different protection schemes can also be realized by mapping the physical pages behind an endpoint buffer to virtual address ranges with different page protections.

3.3 Messages

Messages contain a connection ID (which implicitly identifies the endpoint), control information consisting of an operation and some operands, and data. Each operand is an address, immediate data, or the name of some destination state. Addresses are encoded as an offset from the destination endpoint. The offset is essentially a network logical address that insulates the sender and destination from the addressing details (e.g. address space size, virtual to physical mappings, and page size) at the other. This separation promotes modularity and accommodates node heterogeneity. Furthermore, an offset typically does not need the full dynamic range of a virtual or physical address and thus can be encoded in fewer bits within a message.

3.4 Operations

The simplest operations are pure sender-based direct read and write data transfers. For a direct write, the sender specifies the source data by its offset from the source endpoint base and the destination location by the offset from the destination endpoint base. For reads, the source sends a message with a direct write request to the destination, along with the offset in the destination and the deposit offset in the source (and a reply connection if the connection is not duplex).

To enable operations which are a function of both sender and destination state, the destination end of each connection has some state which message operands can name. To simplify matters (and to foreshadow our implementation in Section 6), we assume this state is contained in specially addressable locations which we call "address registers". (This state could also be held in general memory locations.) Thus in the full model, message actions are a function of an operation specified by the sender, operands representing sender state, and the contents of the address registers.

The operations must be simple enough to complete in one cell time without host processor or operating system intervention. Figure 2 shows an example set of such primitive operations. The address generation subset calculates the effective address (effaddr) at which to read or write for data transfers. Indirection is useful for queue manipulation and more generally for isolating sender and destination. (<X> denotes the contents of location X.) The conditional

Address generation	effaddr= *operand*	(direct addressing)
	effaddr= $< addreg_i >$	(indirect addressing)
	effaddr= $< addreg_i > + operand$	(indexed addressing)
Register operations	$addreg_i \leftarrow operand$	
	$addreg_i \leftarrow$ **unary-op** $\{ < addreg_j >$ or *operand* $\}$	
	$addreg_i \leftarrow < addreg_i >$ **binary-op** $\{ < addreg_j >$ or *operand* $\}$	
Conditional operations	if ($< addreg_i >$ **compare-op** *operand*) then generate interrupt at end	
	if ($< addreg_i >$ **compare-op** $< addreg_j >$) then generate interrupt at end	

Figure 2: Example set of primitive operations

operation subset provides the primitives for conditional interrupts to implement delayable or immediate actions. Finally, the register operation subset allows manipulation of registers for address generation (e.g. postincrementing) and conditional interrupts (e.g. interrupt masks). The message operation controls whether a read or write occurs (or otherwise), the primitives selected, and their order. (The conditional test may occur at any time but the interrupt occurs at the end of the compound operation.)

The few primitive operations in this example set allow a rich set of powerful and flexible compound operations. For example, a store indirect with postincrement can be synthesized with an indirection followed by a register operation:

$< addreg_i > \leftarrow MSG$
$addreg_i \leftarrow < addreg_i > + operand$ (Or $addreg_i \leftarrow < addreg_i > + < addreg_j >$)

Done on a per-cell basis, this amounts to DMA with stride equal to the increment value. However, note that varying $operand/< addreg_j >$ yields variable strides. As another detailed example, we can synthesize priority queueing and interrupts as follows:

$< addreg_p > \leftarrow MSG$
$addreg_p \leftarrow < addreg_p > + < addreg_s >$
`if (`*operand* `greater than` $< addreg_i >$ `)` generate interrupt at end
$addreg_i \leftarrow < addreg_i >$ `bitwise-or` *operand*

operand indicates the priority of the message, $addreg_p$ points to the end of the queue to which this priority message should be added, $addreg_s$ contains the size of MSG, and $addreg_i$ holds the priority level at the destination[2] ($p \neq i \neq s$). The message specifies *operand* and register indices p, i, and s. Complex compound operations like this priority queueing may require multiple compound operations. For example, two compound operation messages would be required in this case if the destination executes one register operation per message.

In a variation of this priority queueing example, we could append messages to one of several different queues without generating interrupts and maintain a bit vector of non-empty queues. This mechanism can be implemented in the same way as priority-based interrupts, but with the most significant bit of $addreg_i$ set to block interrupts.

[2]This assumes fewer queues than bits in $addreg_i$.

As other examples, various atomic operations such as fetch-and-increment, read-modify-write, and compare-and-swap can be implemented by devoting one or more of the address registers for the target location and using the register operations for incrementing and comparing. We can also implement barrier synchronization this way. When a process reaches a barrier point, it toggles a bit in a specified address register of all processes in the "barrier set" and then waits for a conditional interrupt when all the bits are set (or cleared).

3.5 Protection and Isolation

There are two levels of protection: access to and from the network is authorized via connection IDs and the source and sink of endpoint data is authorized by virtual memory mapping and page protections. Together with address register indirection, these mechanisms provide protection and isolation between sender and destination manipulation of endpoints. Towards the destination-based end of the addressing spectrum, there is also the need for protection and isolation between sender and destination manipulation of the address registers. For example, in the priority queueing example in Section 3.4, many senders might send messages to the same endpoint within the operating system. Then it would be necessary for the integrity of the priority queueing (and possibly for the operating system) that a sender not be able to overwrite a queue pointer or the message size. We may also not want a sender to be able to read any of this information. Finally, the destination needs to be able to prevent unwanted interrupts, such as from malicious or errant senders.

To allow a destination to control the information a sender can read and modify, we add address register protection. We assume that each address register can be marked as readable and/or writable by the sender, or accessible only by the destination. This adds protection but does not increase isolation since the sender must still name all the operands. A sender could still give inconsistent operands (e.g. an operand priority and priority queue that do not match) or specify the wrong registers. Also, a malicious sender could still force interrupts at the destination. To solve these problems, we need to isolate operand names to the destination.

To implement this classic destination-based addressing, the destination could decode the operation to find the destination operands or simply interrupt the host processor. The latter option usually incurs significant overhead but can be useful as a flexible escape mechanism. For the former option, we take the following approach to minimize complexity. We extend the hybrid deposit model to allow the operation in a message to be replaced by a pointer to an "instruction" comprising an operation and operands in the destination. This allows the destination operation to directly reference destination operands (in the address registers) without the sender naming those operands, thereby isolating the destination state from the sender. The destination can refuse all non-instruction pointer style messages (on a per connection basis) to give isolation. However, messages can still provide immediate operands and name destination operands, though the destination may choose not to use these operands (as necessary to maintain isolation). This extension is merely another way to name the operations and operands at the destination; there is no change in the set of primitive operations at the destination.

Now, to implement the priority queueing example given earlier, the sender only needs to

specify the instruction pointer for priority enqueueing, the priority, and the data.

Although the instruction pointer approach may appear to subsume the direct operation approach, it really just presents a different cost/benefit point. Some of the increased costs of the instruction pointer approach are a setup phase — the storage of any destination operands and conveyance of an appropriate instruction pointer back to sender — and increased complexity on the part of the destination.

3.6 Summary

Combining sender and destination state in determining message action makes the hybrid deposit model powerful and flexible. We have the full superset of capabilities of the conventional, destination-based approach and sender-based addressing. We have the flexibility to vary the mix of sender and destination information — on a per message basis — to accommodate different requirements on what the sender knows, or alternatively, different requirements on the isolation of knowledge between sender and destination. We can form compound operations from primitive operations. Finally, we can form even more complex operations by combining the result of multiple compound message operations. The hybrid deposit model is more than end-to-end DMA since consecutive messages can be stored in non-sequential locations and non-data operations like conditional interrupts and register operations are supported.

Different application areas are likely to use the hybrid deposit model in different ways. In parallel computing, the endpoint buffers are likely to be large and applications will probably mostly use sender-based addressing, reflecting the cooperation and trust placed in application "partners" on each node. In distributed computing, the endpoint buffers are likely to be small and applications will probably mostly use indirection for isolation and protection. Taking this to the extreme, the hybrid deposit model can model the conventional approach to networking by causing an interrupt on every message (or by having an endpoint within the kernel and using conditional interrupts). In real-time computing, the emphasis will likely be on using conditional interrupts to control asynchronous event delivery. The hybrid deposit model is flexible enough to span all these application areas.

4 Specializing to ATM

In the rest of this paper we specialize the hybrid deposit model to ATM networks — it can certainly be specialized to other networks as well.

Figure 3 shows an example cell format (one of many possible variants) that we will use throughout the rest of the paper. The connection number is encoded in the VCI/VPI field (not shown) in the header. The payload is divided into 32 bytes of data — a size selected to match memory and cache (sub)block sizes — and 16 bytes of control. The opcode/instruction pointer field either directly specifies the operation to be performed at the destination or provides a pointer to the operation and operands at the destination; operand is a 32 bit immediate source operand (offset or data). Destination operands are specified via three separate register indices encoded in the index field. Reads and writes occur in 32 byte blocks. The mask field can be

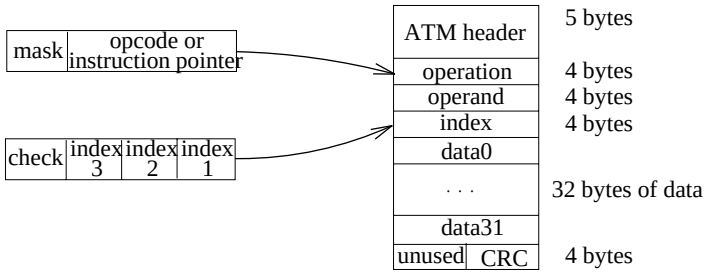

Figure 3: Example format of 53 byte ATM cell for hybrid deposit model

used to deselect the reading or writing of 4 byte words within such a block. (Bit i in the mask controls whether data word i is read or written.) This feature is useful to update a location without changing the values (e.g. variables) in neighboring locations in a block. The check field contains a checksum over the prior control fields so that decoding of these fields can begin before the entire cell arrives.[3] The two byte CRC covers the data field. To take advantage of CRC hardware for AAL5 format cells [Onv94], the CRC field could be extended to 4 bytes and cover the entire payload.

The example in Figure 3 is for a write. For a read request to a remote node, the data section contains control fields for the read reply message.

There is also a multiple cell message format for block transfer. In this format, the first cell is a "control" cell in the write format in Figure 3 and the following cells are standard AAL5 cells. To avoid complexities involving cell boundaries and the trailer (length and CRC) in the last AAL5 cell, all block transfers are multiples of 16 bytes.

5 Software Implementation

To experiment with the hybrid deposit model, especially with different protocol variations and applications, we built a software implementation of the hybrid deposit model described in Section 3.[4] We also wanted to understand the implementation alternatives and performance limits on a stock machine with a primitive network interface to help build a case for hardware assistance in the network interface. In this section we give a quick overview of this implementation and present some preliminary experimental results.

The implementation hardware platform is a DECStation 5000/240 with a 140Mbps Fore Systems TCA-100 ATM interface card. The operating system is Mach 3.0. We modified the Mach kernel to send a cell via an illegal instruction trap, partly optimized the ATM interface interrupt path, and added hybrid deposit emulation in the kernel. This is similar to the software implementation described in [TLL93] except they modified Ultrix and only support simple remote read and write (using only direct addressing). We added more functionality

[3] An 8 bit checksum for 11 control bytes gives more protection than the 10 bit CRC in AAL3/4 cells.

[4] Conditional interrupts have not been implemented yet.

(address indirection and register operations) and modified the cell format (to use 32 bytes instead of 40 bytes of data) in order to be a better match for host computer systems.

We measured the round trip latency for a 32 byte remote write with two DECStations. This is the time to send 32 bytes from user level on the source to user level on the destination, discover via polling that the message arrived, and send the same 32 bytes back to the sender. On the first such cycle, the round trip latency was greater than 1msec, due to TLB and cache misses. For back-to-back connected workstations (i.e. no switch), the average round trip time on successive cycles was 49μsec, yielding a best case one way send to receive time of 24.5μsec. The best case remote read time was 45μsec. Connecting the workstations to our Fore Systems ASX-100 ATM switch added about 5.5μsec per switch transit to these numbers.

Since the TCA-100 ATM interface does not have DMA, all accesses to the interface to send and receive cells must use programmed I/O. Unfortunately, programmed I/O is quite slow on the DECStation TURBOchannel I/O bus resulting in nearly one third of the latency.

For 155Mbps, we estimate that point to point data transfer latencies in the 10 to 20μsec range are achievable using next generation stock hardware, conventional interface cards with DMA, and software implementation. Using Fore's ATM switch as a guide, 15μsec to 26μsec is a reasonable range in a uncongested single switch LAN.

6 Hardware Implementation

While the software implementation in Section 5 shows that stock hardware can get quite low latencies, this comes at the cost of host processor load and bandwidth. Furthermore, the latency is highly dependent on hard-to-control factors such as cache and TLB misses and page faults. The worst case can be 10 times the best case due to memory system (mis)behavior. Predictable latency is important for real-time systems. Consequently, hardware implementations — which can achieve constant low latency and high bandwidth with minimal loading on the host — can be useful even at 155Mbps. Such hardware is essential at 622Mbps for both low latency and high bandwidth. While software implementations might be adequate to achieve 10μsec latency in first generation 155Mbps ATM LANs, they will not be adequate to achieve latencies under 3μsec in second generation 622Mbps ATM LANs.

We present an architecture called DART which implements the hybrid deposit model described in Section 3. Figure 4 shows a general block diagram of the common DART components. This paper focuses on the Connection Table, Operation Logic, and Send Registers. The Receive and Send Control blocks are finite state control machines. The remaining function blocks are either rather standard for ATM interfaces or uninteresting (for this paper).

6.1 DART Overview

The Connection Table contains state for each active connection (incoming and outgoing) as shown in Figure 5. Each entry contains an endpoint number, address register information (explained later), some connection state, and a reply connection number. The endpoint number indexes into the Endpoint Table which contains the buffer base and bounds information

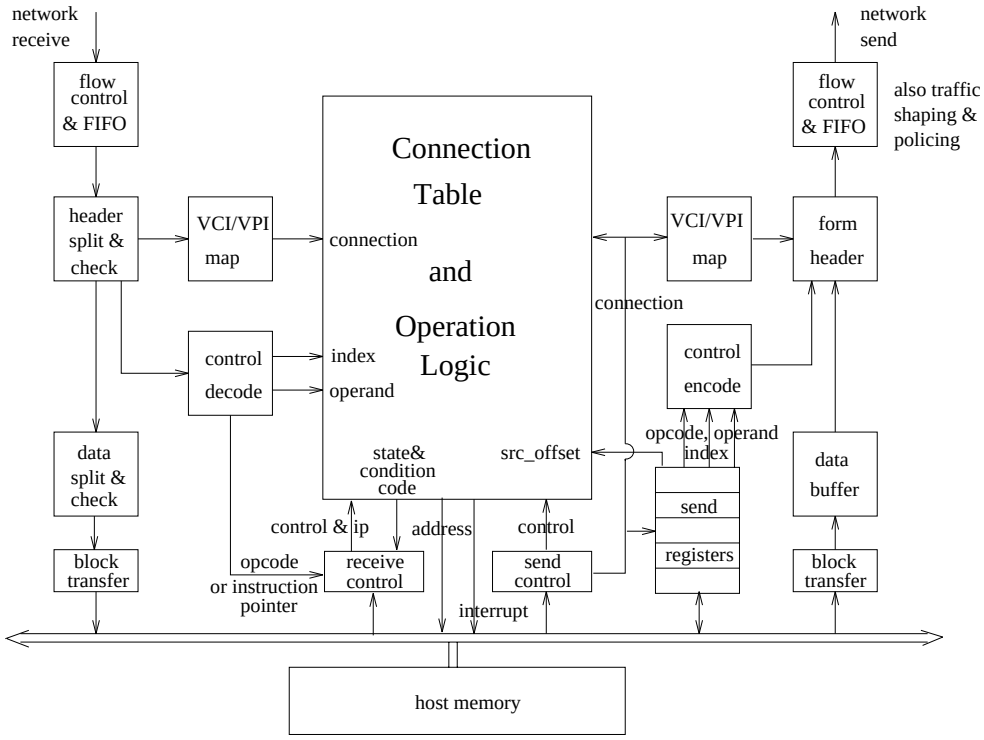

Figure 4: DART block diagram

for each endpoint buffer. The endpoint information is in a separate table so that multiple connections to the same endpoint buffer can share the information. Finally, the reply connection indicates the connection number to use to reply in responding to an incoming message. The Connection Table is mapped into uncached processor address space and managed by the operating system.

The Send Registers are a set of six registers for each active outgoing connection. These registers are used for forming send messages. Each connection's send registers can be mapped (and marked uncached), via an operating system call, into the first six locations in a page in the application address space. Thereafter the application can access the send registers without operating system interaction.

The Operation Logic in Figure 5 performs protection, mapping, and the primitive operations listed in Figure 2. For simplicity, we make the following initial assumptions: Address register operations are restricted to at most one address register read and write operation per cell; endpoint pages must be pinned in physical memory while an endpoint buffer is active; remote reads must be handled by the host processor (via an interrupt); and the instruction pointer message variant is not supported. We also omit latches and control wiring for clarity.

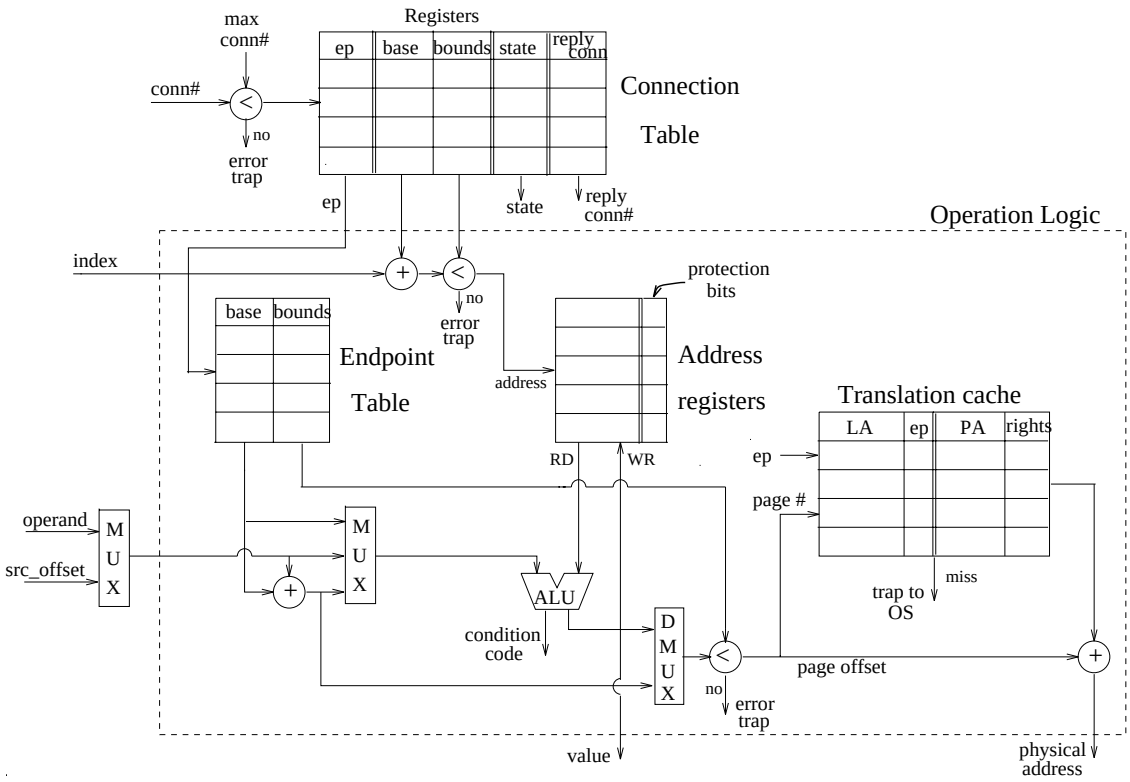

Figure 5: Connection Table and Operation Logic

6.2 DART Receive Side

Incoming messages index into the Connection Table using a connection number derived from the cell VCI/VPI. Each connection has a number of local memory locations for address registers. This introduces the awkwardness of a separate name space, but avoids main memory accesses in the critical path (for indirection). Each connection has a number of contiguous locations to form a register "window". For convenience and flexibility, we allow each connection to dynamically allocate the window size at connection set up time. The address register base and bounds fields in the connection table entry point to the beginning and end of this window respectively. This scheme allows the overlapping and nesting of register windows to effect different sharing and protection (as described for endpoints in Section 3.2). The protection bits on each address register control read, write, and access privileges as described in Section 3.5.

Because the endpoint pages are pinned in this architecture, the Endpoint Table contains the physical address of the buffer base. However, since physical pages are not necessarily allocated contiguously, we use a TLB-like address translation cache to contain mappings from the logical address of the endpoint base address + offset to the appropriate physical address. Note that the TLB must match on bits identifying the endpoint as well as on the physical

address (PA) since multiple endpoints may have the same PA but different protection. A TLB miss causes an interrupt to the host processor. This allows full software flexibility in managing the storage of endpoint mappings. One way to reduce the number of TLB misses is to use a hybrid scheme which stores the first N endpoint buffer page mappings directly in the Endpoint Table and manage any overflow mapping entries with the TLB. A small number, like $N = 2$ is probably sufficient for most endpoints.

To implement a multiple cell format for block data transfers, data cells following an initial "control" cell (as described in Section 4) are deposited in successive memory locations after the data in the control cell. The state field in the Connection Table entry indicates whether an arriving cell for that connection should be interpreted as a control cell or a data cell.

The Address Registers, and TLB are all mapped, via additional data and control paths, into the processor address space so the operating system can manage their contents.

6.3 DART Send Side

Both control information and data must be provided to send a cell. The control information comes from the set of Send Registers associated with an outgoing connection and mapped into a known location in the application address space. The first three registers in this set are the control information — the opcode, operand, and index — for the cell. The data comes from the endpoint associated with the connection. The fourth register, named "go" controls the initiation of a send.

The send side reuses the Operation Logic to map from the application address space to physical endpoint addresses, i.e. we multiplex the Operation Logic between the receiver side and the sender side. To send a block of data at *src_offset* from an endpoint base, we simply write *src_offset* into the "go" Send Register for the appropriate connection attached to that endpoint. This write causes a data block at the endpoint offset indicated by *src_offset* (32 byte block aligned) to be read and composed into a cell with the control information in the opcode, operand, and index registers and sent via the associated connection.

The set of Send Registers for each connection also contains a size register and a mode register. The size register controls the number of 16 byte data blocks (minimum of 2 blocks) sent starting from the offset written into the "go" Send Register. This register is set to 0 when the last cell has been sent to the flow control and traffic shaping unit. The mode register enables two variants of the send procedure just described. The first is a shortcut: the operand is taken from the value actually written to the "go" register. The second causes an exception if an attempt is made to send when the status field is non-zero.

6.4 Exception processing

Exceptions arising due to error traps, TLB misses, and unimplemented operations cause an interrupt to the host processor. (Error traps can also be configured to discard the offending cell without causing an exception.) Any state operated on by the cell, e.g. write to address registers, does not get updated until after the point of the last possible exception point for

that cell. We retain the cell in the input FIFO and block processing of further cells from that connection until re-enabled by the host processor: cells are only removed from the input FIFO when a cell "commits" after the last exception point. (We assume per connection buffering and flow control at the destination.) This in turn might cause the flow control mechanism for that connection to be invoked. Cells belonging to other connections can be processed once the exception condition is saved (even if these connections share the same endpoint as an exceptioned connection). We cannot process cells belonging to the exceptioned connection, even if they are not affected by the exception, since some applications may depend on the per-channel ordering guarantee of ATM cells.

Cell processing can also be blocked globally across all connections. The operating system uses this feature to get atomic access to DART state.

6.5 Relaxing the assumptions

To allow the execution of up to four primitive operations per message (one address generation, two register operations, and one conditional)[5] we clock the Operation Logic through multiple primitive operations per message, feeding back immediate values as necessary via the "value" path shown in Figure 6. A main opcode controls the selection and ordering of the primitive operations. Example opcodes are read, read multiple, write, write multiple, and software exception which causes an interrupt to the host processor. The instruction format allows up to three different register operands to be named in addition to an immediate operand. To retain the simple roll-back exception model in the face of multiple register operations, address register writes are cached and only written back after the cell processing commits.

Removing the restriction on pinning endpoint pages is straightforward: the Endpoint Table now contains virtual addresses and the TLB maps from virtual addresses to physical addresses. This enhancement also introduces a new category of exceptions: page faults from references to paged out endpoint pages. These are treated as another class of exceptions and are serviced by the host processor (which maintains the main virtual mapping tables). The operating system is now responsible for keeping the mapping information in DART consistent with the main memory state.

Handling remote reads without the interrupting the host processor just requires a more complicated receive controller.

To support the instruction pointer message variant, we modify the Connection Table and add an Instruction Memory on the receive side as shown in Figure 6. The Connection Table now contains a base and bounds entry, similar to that for address registers, for access to the Instruction memory. The instruction pointer (ip) from the receive control logic and the ipbase and ipbnds information from the Connection Table combine to index into the Instruction Memory. To keep the scheme simple, each instruction is composed of an operation and operands in exactly the same format as in an ATM cell (as in Figure 3). The operation controls from which location — the instruction memory in the destination or the message — operands and indices are taken. Protection bits, like those for the address registers, allow the destination

[5] So we can do priority queueing and interrupts.

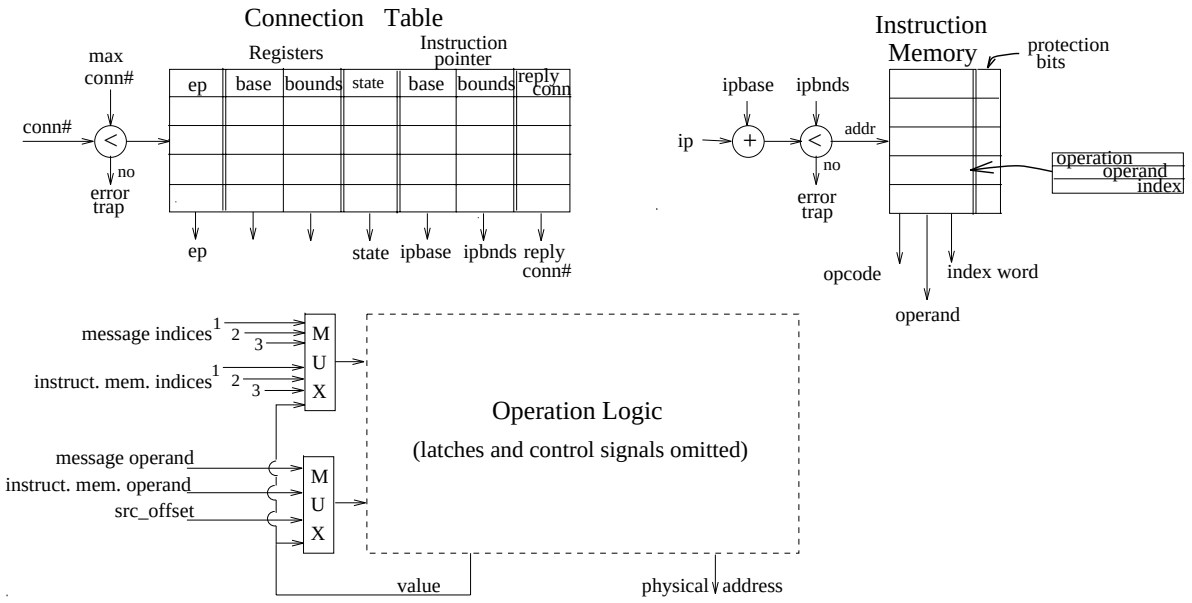

Figure 6: Modifications for multiple primitive operations and instruction pointer format

to control which instructions serve as entry points to the sender. One very useful addition is a sequencer to step the destination through several instructions per cell. It is tempting to add further enhancements, such as conditional sequencing operations, but we avoid them to keep the interface simple: more complex functionality can be obtained by trapping to the host processor.

6.6 Status

At this point we have only completed a high level design of DART. We omitted some of the detail (such as data masking) from the architectural sketch just presented; many more details and implementation options require further evaluation. Some of the major open issues are programmable control for destination operand decoding, the division between hardwired and microcontrolled (e.g. using a small microprocessor) operations, and the duplication of the Operation Logic for sender and receiver. We are planning to interface DART to the PCI bus.

The only difficult timing constraint is completing whatever bus operations are required for data transfers before the control information in the next cell arrives. The address generation, register, and conditional operations can be overlapped with the data and CRC arrival which takes 1.8μsec at 155Mbps and 450nsec at 622Mbps. Since register operations take one cycle to access each register plus one cycle to writeback, the worst case of an address operation, two binary register operations, and a conditional operation on registers takes nine operation logic cycles. This yields a generous clock period of 200nsec at 155Mbps and 50nsec at 622Mbps.

7 Conclusions and Further Work

We presented a new model for low overhead communication. This hybrid deposit model increases the role of the sender in order to simplify the role of the destination. Using both sender-supplied information and destination information, messages are routed so as to minimize the impact on the host processor: data messages are deposited directly in memory, delayable action messages are queued for later processing when convenient, and immediate action messages are sent directly to the host processor. Moreover, the sender and destination information can be combined in a flexible way to span the spectrum from purely sender-based communication, as in some parallel computing systems, to conventional, purely destination-based communication. Thus, our hybrid deposit model is applicable across a wide range of applications (as well as a wide range of networks) in parallel, distributed, and real-time computing.

We specialized this hybrid deposit model to ATM LANs and exploited the synergy between the relatively small, fixed size data units in ATM and common memory block sizes. Our preliminary results show that a purely software implementation on stock workstation hardware can do fairly well, to a point. However, hardware is required to get consistent low latency, high bandwidth, and reduced processor load. We sketched an architecture called DART which hardware demultiplexes incoming cells directly to where they are required, on a per-cell basis. DART provides a small number of demultiplexing primitives in hardware, along with some simple ways to perform compound multiplexing operations. More complex operations are handled via software exception to the host processor.

The per cell processing architecture of DART is similar in principle to the message-driven processor (MDP) [De87]. However, we use DART in a filtering role for depositing messages, rather than for direct computation and we provide full protected multiuser communication. The closest related work we know of for LANs are Hamlyn and Axon. Hamlyn [Wil92] adopts a similar emphasis on increasing the participation of the sender to minimize the required functionality at the destination. However Hamlyn differs in some major ways: it supports only direct addressing, one delayed action queue per node (there can be any number in DART), and it pins endpoint pages. Like DART, Axon [SP90] has self describing packets for direct deposit at the destination. However, Axon lacks hardware support for flexible sender and destination-based addressing, hybrid interrupt control, and register operations.

To date we have developed the model and designed plausible implementations. Now we intend to turn our attention mostly to the other half of the hybrid deposit "hypothesis": examining how higher level protocols such as TCP/IP, services such as RPC, and applications can benefit and what services they require in an network interface. We plan to use our existing software implementation as the basis for this work.

References

[AI87] Arvind and R. Ianucci. Two Fundamental Issues in Multiprocessing. In *Proc. of DFVLR – Conf. on Parallel Processing in Science and Eng.*, June 1987.

[Be94] M. Blumrich and et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Intl Symposium on Computer Architecture*, April 1994.

[BP93] D. Banks and M. Prudence. A High-Performance Network Architecture for a PA-RISC Workstation. *Journal of Selected Areas in Communications*, pages 191–202, February 1993.

[Dav93] B. Davie. The Architecture and Implementation of a High-Speed Host Interface. *Journal of Selected Areas in Communications*, pages 228–239, February 1993.

[De87] W. Dally and et al. Architecture of a Message-Driven Processor. In *Intl Symposium on Computer Architecture*, 1987.

[DLM94] C. Dubnicki, K. Li, and M. Mesarina. Network Interface Support for User-Level Buffer Management. In *Parallel Computer Routing and Comm. Workshop, Univ. of Washington*, May 1994.

[DP93] P. Druschel and L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proc. of the Sympos. on Operating System Principles*, December 1993.

[Ke94] P. Keleher and et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of Winter Usenix Conf.*, January 1994.

[NPA92] R. Nikhil, G. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Intl Symposium on Computer Architecture*, pages 156–169, May 1992.

[Onv94] R. Onvural. *Asynchronous Transfer Mode Networks: Performance Issues.* Artech House, 1994.

[Se93] J. Subhlok and et al. Programming Task and Data Parallelism on a Multicomputer. In *Proc. of ACM Sympos. on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.

[Se94] T. Stricker and et al. Decoupling Communication Services for Compiled Parallel Programs. Technical Report CMU-CS-94-139, CMU, 1994.

[SP90] J. Sterbenz and G. Parulka. Axon: A High Speed Communication Architecture for Distributed Applications. In *Proceedings of IEEE INFOCOM*, 1990.

[Spe82] A. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, pages 246–260, April 1982.

[SR91] J. Stankovic and K. Ramamrithm. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3), May 1991.

[TLL93] C. Thekkath, H. Levy, and E. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report TR93-04-03, Dept. of Computer Science, Univ. of Washington, April 1993.

[TS93] C. Traw and J. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *Journal of Selected Areas in Communications*, pages 240–253, February 1993.

[von92] von Eicken et al. Active Messages: A Mechanism for Integrated Communication and Computation. In *Intl Symposium on Computer Architecture*, pages 256–266, May 1992.

[Wil92] J. Wilkes. Hamlyn: An Interface for Sender-based Communication. Technical Report HPL-OSR-92-13, HP Labs, November 1992.

Acknowledgments

MERL colleagues Chia Shen, Richard Waters, John Howard, Hugh Lauer, and Qin Zheng gave helpful suggestions on the interface ideas and paper presentation. John Kubiatowicz of the MIT Lab for Computer Science gave valuable feedback on the design ideas and presentation in an earlier draft. The development of the hybrid deposit model benefitted from discussions with Peter Steenkiste and Thomas Gross at CMU. Takushi Kawada and Vu Le Phan of Mitsubishi Electric helped install Mach 3.0.